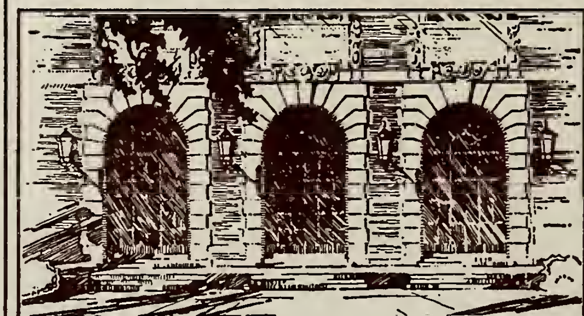


Math

Report No. 453

MACHINE-INDEPENDENT COMPILATION OF PL/I: PASS 1 COMPILATION*

by

Paul Jang-Ching Wang

June, 1971

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

Report No. 453

MACHINE-INDEPENDENT COMPILATION OF PL/I: PASS 1 COMPILATION*

by

Paul Jang-Ching Wang

June, 1971

Department of Computer Science
University of Illinois
Urbana, Illinois 61801

*Submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science in the Graduate College of the University of Illinois, 1971.

MACHINE-INDEPENDENT COMPILATION OF PL/I: PASS 1 COMPILATION

BY

PAUL JANG-CHING WANG
B.S., Taiwan Cheng Kung University, 1965

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1971

Urbana, Illinois

ACKNOWLEDGMENT

The author would like to express his sincere appreciation to his advisor, Professor H. G. Friedman, for his guidance and his invaluable suggestions to make this thesis complete. Mrs. Freda Fischer must also be thanked for her assistance in debugging the EOL program.

Thanks are extended to Miss Joyce Vaughn for typing this thesis. Finally, the author would like to express his gratitude to the ILLIAC IV project for its financial support.

TABLE OF CONTENTS

Page

1. INTRODUCTION. 1

2. INTERMEDIATE LANGUAGE 8

2.1 The Types of PL/I Statements. 8

2.2 Conventions 9

2.3 General 10

2.4 Definitions and Examples of Intermediate Language 11

2.4.1 The Assignment Statement. 12

2.4.2 The BEGIN Statement 14

2.4.3 The CALL Statement. 15

2.4.4 The DECLARE (DCL) Statement and the DEFAULT Statement 16

2.4.5 The DISPLAY Statement 16

2.4.6 The DO Statement. 17

2.4.7 The END Statement 19

2.4.8 The ENTRY Statement 19

2.4.9 The EXIT Statement. 20

2.4.10 The FORMAT Statement 20

2.4.11 The GET Statement. 20

2.4.12 The GO TO (GOTO) Statement 22

2.4.13 The IF Statement 23

2.4.14 The NULL Statement 24

2.4.15 The PROCEDURE (PROC) Statement 25

2.4.16 The PUT Statement. 26

2.4.17 The RETURN Statement 28

2.4.18 The STOP Statement 28

Page

2.4.19 The Other Statement. 29

2.4.20 DATA-ATTRIBUTE-LIST. 29

2.4.21 FORMAT-LIST. 31

2.4.22 DATA-LIST. 33

3. STATEMENT RECOGNIZER. 34

3.1 Expression Translator 36

3.1.1 Definition of Reverse Polish Expression 36

3.1.2 Conversion of Infix Expression. 37

3.2 Recursive Procedures. 45

3.3 External Procedures 51

3.3.1 Procedure ST.REKNZER.C. 51

3.3.2 Procedure STATEMENT.RECOGNIZER. 51

3.3.3 Procedure ERROR 53

3.3.4 Procedure READ. 54

3.3.5 Procedure DICT. 54

3.4 Internal Procedures 54

3.4.1 Generator Group 54

3.4.1.1 Procedure ASSIGN. 54

3.4.1.2 Procedure BEGIN 55

3.4.1.3 Procedure CALL. 55

3.4.1.4 Procedure DECLARE (DCL) 55

3.4.1.5 Procedure DEFAULT 55

3.4.1.6 Procedure DISPLAY 55

3.4.1.7 Procedure DO. 56

3.4.1.8 Procedure END 57

Page

3.4.1.9 Procedure ENTRY 57

3.4.1.10 Procedure EXIT 57

3.4.1.11 Procedure FORMAT 57

3.4.1.12 Procedure GET 57

3.4.1.13 Procedure GO 58

3.4.1.14 Procedure IF 58

3.4.1.15 Procedure NULL 58

3.4.1.16 Procedure PROCEDURE. 58

3.4.1.17 Procedure PUT. 58

3.4.1.18 Procedure RETURN 62

3.4.1.19 Procedure STOP 62

3.4.1.20 Procedure CLOSE and So On. 62

3.4.1.21 Procedure ELSE 62

3.4.2 Utility Group 62

3.4.2.1 Procedure READB 62

3.4.2.2 Procedure DICTB 63

3.4.2.3 Procedure VARIBL. 63

3.4.2.4 Procedure DLIST2. 63

3.4.2.5 Procedure DLIST3. 63

3.4.2.6 Procedure ATRIBU. 63

3.4.2.7 Procedure ASSIGNMENT. 64

3.4.2.8 Procedure POLISH. 64

3.4.2.9 Procedure ALPHAOPT. 64

3.4.2.10 Procedure INTEGER? 64

3.4.2.11 Procedure HANDLEASSIGN 64

Page

3.4.2.12 Procedure LEGAL?ID 65

3.4.2.13 Procedure FOMAT. 65

3.4.2.14 Procedure COND.PREFIX. 65

3.4.2.15 Procedure POLI 65

3.4.2.16 Procedure DEBUG. 65

4. CONCLUSIONS AND SUGGESTIONS 66

BIBLIOGRAPHY. 67

APPENDIX

A. THE TIMETABLE OF THE PROPOSED PL/I COMPILER . . . 69

B. WHEN TO CALL PROCEDURE DICT 74

LIST OF FIGURES

Figure Page

1. A typical two-pass compiler organization........... 6
2. Structure of the compiler program.................. 34
3. Structure of the procedure STATEMENT.RECOGNIZER.... 35
4. Structure of the Generator Group................... 35
5. Structure of the Utility Group..................... 36
6. Flow diagram of the procedure STATEMENT.RECOGNIZER. 52
7. Flow diagram of the procedure IF................... 59
8. Flow diagram of the procedure DOBEGIN.............. 61

1. INTRODUCTION

As far as money is concerned, it is very obvious that small or medium scale computers are more popular than large scale computers. The machine cost of the former is always much less than that of the latter. Most of the organizations (such as companies, universities, etc.) just do not have enough money to buy or lease large scale computers. These two factors do affect the investment policy in industry society.

PL/I is a programming language for all seasons. It is the new efficient and powerful generation of the marriage between commercial and scientific languages. Because a PL/I compiler itself occupies too much computer memory, it is difficult to make PL/I available to small or medium scale computer users. It would be a tremendous advantage to make PL/I available on this kind of computer. The solution to the problem mentioned above is to compile the source PL/I program on a convenient large scale computer and to run the object code (i.e., the output from that large scale computer) on the specific small or medium scale computer. PL/I has drawn features from COBOL, FORTRAN, ALGOL, and other languages. It represents a great step forward to guide the activities of a computer and has been generated based on the experience accumulated with a wide variety of previous compilers. This enhances more characteristics for PL/I. Compilers for previous high-level languages were so designed that the programmer had to qualify every identifier, every option, and every specification. In sharp contrast, the PL/I compiler takes an action the programmer has failed to take. This property

is known as 'default action.' When the user is throughly familiar with the default action which the compiler will take and if this is in line with the program as intended, he has a powerful mechanism for saving programming time and computer memory. An effort was attempted to organize the language structure in such a way that the beginner with no prior experience can code effective PL/I programs by simply learning a very small subset of PL/I statements and attributes. PL/I operates at a higher level than was generally available. It has several features which are seldom found in other languages such as array assignment statement, structure assignment statement, multiple closure, do group, etc. Thus, here at the University of Illinois, a small group started to design a PL/I compiler. This project developed into the following steps.

1. Select a subset of PL/I for which to implement the compiler for the present time.
2. Choose a language in which to write the compiler program.
3. Design the intermediate language.
4. Design and implement the first pass on IBM System/360.
5. Design and implement the second pass on IBM System/360.

The last step was to be left to future efforts.

A table of PL/I statements and another of PL/I attributes were set up. These two tables formed the timetable of the proposed PL/I compiler. (See Appendix A.) The subset of PL/I in the first version was carefully selected so that all the statements and attributes in that set were the most practical and the most fundamental features.

There are four languages involved in the compilation process.

1. The source language, i.e., PL/I.
2. The language in which the compiler program itself may be written and described.
3. The intermediate language, i.e., the first object language.
4. The assembly language, i.e., the second object language.

In a search for a good language to use to write the compiler, the following goals should be considered.

1. It should process lists, character strings, and stacks efficiently.
2. Its programs should run quite efficiently.
3. It could be easily implemented on most computers.

It was decided that EOL was the best language to code the proposed compiler because EOL satisfied all the goals mentioned above.

EOL is a simple low-level language for character string manipulation. It has been designed mainly for compiler writing and symbolic and algebraic manipulation. The development of EOL started in 1965 at the Institute for Mathematical Machines in Warsaw, Poland. The initial EOL-1 and EOL-2 versions were elaborated there. Based on these two previous versions an improved EOL-3 version has been developed and implemented on IBM 7094 and System/360 at the University of Illinois. EOL-3 facilitates one input device and three output devices. It offers thirty-two stacks for string processing and two files for mass storage.

EOL is a universal but relatively concise language equipped with only a limited set of statements. It has the facilities to handle most of the typical operations used in compilers. Instructions are classified into eight groups: stack, input, output, file, jump, table, macro, and code. Declarations are classified into five groups: switch, table, entry, external, and index. The switch jumps (go or call according to name or index switch) are very desirable features for the compiler writer. They are included in EOL. It is possible to manipulate instruction address stack in order to direct the control to a proper point by applying the instruction EXCHANGE. Instruction CODE allows part of an EOL program to be written in machine language.

The pass 1 output will be stored in an EOL file and will be expressed in intermediate language code. A better design of intermediate language will lead to a more efficient pass 2 because a string of information has been represented by the intermediate language code. Thus, an intermediate language code which is represented in a minimum number of characters and gives a maximum amount of information can save much more time for pass 2 while pass 2 rescans the intermediate language code. The definition of intermediate language for each PL/I statement will be given in Chapter 2.

The input to pass 1 is PL/I code and the output from pass 1 is the intermediate language code. The intermediate language code accompanied by name tables is in turn the input to pass

2 and the output from pass 2 is now assembly language code. (See Figure 1.)

It was proposed that pass 1 should do as much as possible. In pass 1, the semantic actions construct name tables containing information about block structures, declaration types, and attributes, and map the source program into intermediate language code which is composed of codes, identifiers, variables, names, operators, and operands.

Figure 1 shows a typical two-pass compiler organization. The upper half is pass 1 which is composed of five parts. The statement recognizer is the largest and the most important part among these five parts. It calls procedure READ to get a statement and calls procedure DICT whenever an identifier is encountered. It analyzes the statement and transforms the statement into an intermediate language code which is an EOL file record ended by a semicolon and is stored in EOL file 2. When an error is discovered, statement recognizer calls procedure ERROR which prints out an error message. Usually, it continues to analyze the same statement as long as it can.

The proposed compiler has two passes, but its compiling time is much less than twice because of efficiencies introduced by the intermediate language. There are two basic methods of compiling, the buttom-up and the top-down methods. In order to enjoy more benifits of both methods, the bottom-up method (i.e., operator precedence method) was employed to transform the expressions into Reverse Polish expressions, while the top-down method was used to analyze all syntax-oriented statements.

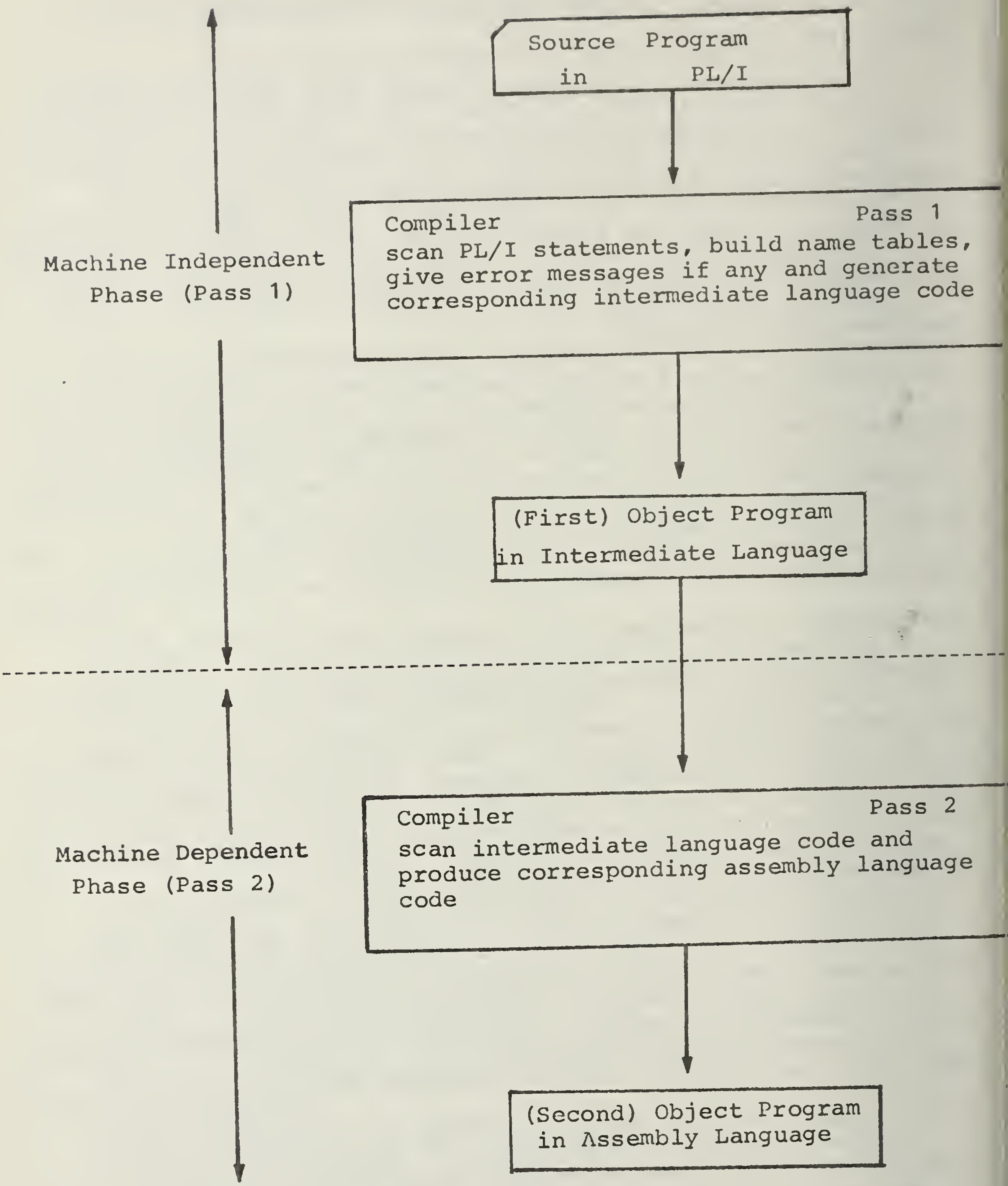


Figure 1. A typical two-pass compiler organization.

All PL/I statements except the assignment statements begin with a keyword such as DO, IF, BEGIN or PROCEDURE and end with a semicolon. The assignment statements have no keywords but they have special leading patterns. In the statement recognizer, a very efficient algorithm was employed. It is able to identify all legal statements without requiring keywords to be reserved. Thus, 'IF (X+Y), THEN, Z = 10;' is a legal assignment statement. The statement recognizer identifies the type of each statement before the parse of that statement is attempted. It first tries to recognize assignment statement by looking for a correct leading pattern which is roughly analogous to X, Y(I+J,2,K), Z=... or X=... etc. If this fails, the leading identifier is matched against a keyword table to determine the statement type.

A detailed description of the intermediate language will be presented in Chapter 2. The statement recognizer and the techniques and the principles used in the statement recognizer are briefly described in Chapter 3. Some conclusions and suggestions are given in Chapter 4.

2. INTERMEDIATE LANGUAGE

Pass 1 stores a file of intermediate language code which is a special representation of the original PL/I program. The introduction of intermediate language is particularly valuable because the proposed PL/I compiler is a multi-pass compiler and it is not necessary to list the source program during pass 2, in which the time to reread the original source program can be completely saved.

2.1 The Types of PL/I Statements

PL/I statements may be classified into nine logical groups: assignment, control, data declaration, error control and debug, program structure, storage allocation, unclassified, output, and input. A digit is assigned to each group for later use.

0-group	contains	all scalar, array, structure, pointer, and label assignemnt statements. (assignment statements group.)
1-group	contains	CALL, DO, EXIT, GOTO, IF, NULL, STOP, RETURN, DELAY, and WAIT statements. (control statements group.)
2-group	contains	DECLARE and DEFAULT statements. (data declaration statements group.)
3-group	contains	ON, REVERT, and SIGNAL statements. (error control and debug statements group.)
4-group	contains	BEGIN, END, PROCEDURE, and ENTRY statements. (program structure statements group.)
5-group	contains	ALLOCATE and FREE statements. (storage allocation statements group.)

6-group contains INCORPORATE statement. (unclassified statement group.)

7-group contains PUT, WRITE, REWRITE, LOCATE, DISPLAY, RELEASE, and FORMAT statements. (output statements group.)

8-group contains GET, READ, FETCH, DELETE, UNLOCK, OPEN, and CLOSE statements. (input statements group.)

There are several statements which belong to both output statements group and input statements group. Because there are more than ten statements in these two groups, they have to be separated into two groups.

2.2 Conventions

The following conventions are used through this chapter.

$|$ means that a choice is to be made.

$\{\ \}$ means that precisely one of the elements within the brackets is selected.

$[\ ]$ means that none or one of the elements within the brackets may occur.

$\{\ \}\ldots$ means that the elements within the brackets may form an arbitrary non-empty sequence.

$[\ ]\ldots$ means that the elements within the brackets may form an arbitrary, possibly empty sequence.

$\underline{xx..x}$, for a list xx..x, means the same list as xx..x with subscripts, expressions or arguments transformed into Reverse Polish form, ',' replaced by one blank, and meaningless blanks deleted.

b̸ means exactly one blank.

2.3 General

All infix expressions in source program are always transformed into Reverse Polish expressions. Any label prefixes (including entry names) will not appear again in intermediate language code since dictionary routine DICT has put them into a name table and reading routine READ has deleted them from the STRING stack. The condition prefixes will be translated as follows:

convert

{(condition-name[,condition-name]...):}...

into

{(condition-code[,condition-code]...):}...

Where condition-code is the corresponding code for condition-name. The one-to-one correspondence between condition-name and condition-code is defined as following:

condition-name	condition-code
UNDERFLOW	00
OVERFLOW	01
ZERODIVIDE	02
FIXEDOVERFLOW	03
CONVERSION	04
SIZE	05
STRINGSIZE	06
STRINGRANGE	07
SUBSCRIPTRANGE	08
CHECK(identifier-list)	09 (identifier-list)
NOUNDERFLOW	10

NOOVERFLOW	11
NOZERODIVIDE	12
NOFIXEDOVERFLOW	13
NOCONVERSION	14
NOSIZE	15
NOSTRINGSIZE	16
NOSTRINGRANGE	17
NOSUBSCRIPTRANGE	18
NOCHECK(identifier-list)	19 (identifier-list)

If condition-name is undefined, then an error message will be printed out and that undefined name will be skipped. For example:

(ill-condition-name,OVERFLOW,ill-condition-name,SIZE
,ill-condition-name):

is translated into

(,01,,05,):

accompanied by error messages.

In general, blank, as a delimiter, is used to replace comma is a list, i.e., ARRAY(B,C) is translated into ARRAY(BȻC). Comma is usually used as delimiter in Reverse Polish form, i.e., B*C is translated into B,C,*.

2.4 Definitions and Examples of Intermediate Language

If e is a PL/I infix expression, POLISH(e) is defined as the Reverse Polish form of e. For example:

If e=A*(B+C)**D-E

then POLISH(e)=A,B,C,+,D,**,*,E,-

POLISH can be considered as a mathematical function. Its domain is the PL/I infix expression and its image (range) is the

corresponding Reverse Polish form. POLISH is really the most important procedure in the statement recognizer.

The general format for the intermediate language code of a PL/I statement except assignment statement is:

[(condition-code[,condition-code]...):]...type-code rest;

Where condition-code was defined in 2.3. The type-code is a two digit integer. The first digit is exactly the digit which was assigned to each logical group in 2.1. Since each group contains no more than ten statements the second digit can be selected from 0 to 9. The rest is the remaining information about that statement.

The definitions of PL/I statements and attributes are mainly based on IBM (1969).

2.4.1 The Assignment Statement

General Format:

$$\left\{\begin{array}{l}\text{scalar-variable}\\\text{pseudo-variable}\end{array}\right\}\left[\begin{array}{l}\text{,scalar-variable}\\\text{,pseudo-variable}\end{array}\right]\text{...=scalar-expression;}$$

$$\left\{\begin{array}{l}\text{array-variable}\\\text{pseudo-variable}\end{array}\right\}\left[\begin{array}{l}\text{,array-variable}\\\text{,pseudo-variable}\end{array}\right]\text{...}$$

$$=\left\{\begin{array}{l}\text{structure-expression[,BY NAME]}\\\text{array-expression[,BY NAME]}\\\text{scalar-expression}\end{array}\right\};$$

```
{structure-variable} [,structure-variable]...
{pseudo-variable   } [,pseudo-variable   ]
                                    ={structure-expression[,BY NAME]};
                                     {scalar-expression             }
```

Intermediate language:

```
{scalar-variable}   [,scalar-variable]...=POLISH(scalar-expression);
{pseudo-variable}   [,pseudo-variable]
{array-variable }   [,array-variable ]...
{pseudo-variable}   [,pseudo-variable]
                                    ={POLISH(structure-expression)[,']};
                                     {POLISH(array-expression)[,']    }
                                     {POLISH(scalar-expression)       }
{structure-variable} [,structure-variable]...
{pseudo-variable   } [,pseudo-variable   ]
                                    ={POLISH(structure-expression)[,']};
                                     {POLISH(scalar-expression)       }
```

BY NAME in source program is replaced by a quotation mark (') in intermediate language code and BYNAME is considered to be BY NAME.

```
Example: (OVERFLOW):LABEL: V,A(I*J+6,I-7)=C*D-6;
           ↓
         (01):V,A(I,J,*,6,+b̸I,7,-)=C,D,*,6,-;

Example: M='MONEY'||'MO/*C*/NE;Y';
           ↓
         M='MONEY','MO/*C*/NE;Y',||;

Example: Z=B CAT C;
           ↓
         Z=B,C,CAT;

         (Where CAT is an alphabetic operator standing for ||)

Example: A=B(C(I+J)*D/K)+E;
           ↓
         A=B(C(I,J,+),D,*,K,/),E,+;

Example: A=B+C,BY NAME;
           ↓
         A=B,C,+,';
```

```
Example: SUBSTR(R,2,3)=(10)STRING;
            ↓
         SUBSTR(Rb̸2b̸3)=(10)STRING;

Example: IF(X+Y),THEN,Z=(A+B1)/C2D*EE-3.0E10;
            ↓
         IF(X,Y,+),THEN,Z=A,B1,+,C2D,/,EE,*,3.0E10,-;

Example: X=104.17.0L+0.13.4.2L*31.0.2L/0.17.0L;
            ↓
         X=104.17.0L,0.13.4.2L,31.0.2L,*,0.17.0L,/,+;

Example: A.B.C=D.E+F.G.H;
            ↓
         A.B.C=D.E,F.G.H,+;
```

2.4.2 The BEGIN Statement

General Format:

```
BEGIN [ORDER|REORDER] [OPTIONS(option-list)];
```

Intermediate language:

type-code	1st field	2nd field	
	0	0	
40	1		;
	2	1(option-list)	

1st field: 0 ⟶ empty
1 ⟶ ORDER
2 ⟶ REORDER

2nd field: 0 ⟶ empty
1 ⟶ OPTIONS

```
Example: Y:BEGIN  REORDER;
           ↓
         4020;

Example: (CHECK(X,Y),SIZE,CONVERSION):(NOOVERFLOW):Y:BEGIN ORDER;
            ↓
         (09(X,Y),05,04):(11):4010;

Example: BEGIN OPTIONS(OP1,OP2,OP3) REORDER;
            ↓
         4021(OP1b̸OP2b̸OP3);
```

```
Example: BEGIN REORDER OPTIONS(OP1,OP2,OP3);
            ↓
         4021(OP1ƀOP2ƀOP3);
```

2.4.3 The CALL Statement

General Format:

```
CALL {entry-expression|generic-name}
[(argument[,argument]...)] [TASK[(scalar-task-expression)]]
[EVENT(scalar-event-expression)] [PRIORITY(scalar-expression)];
```

Intermediate Language:

```
type-code
   10 {entry-expression | generic-name}[(POLISH(argument) [ƀPOLISH
                                       1st field
   (argument)]...)]                       0
                                          1[(scalar-task-expression)]
2nd field                              3rd field
   0                                      0
   1 (scalar-event-expression)            1 (POLISH(scalar-expression))
1st field:     0 ——→ empty
               1 ——→ TASK
2nd field:     0 ——→ empty
               1 ——→ EVENT
3rd field:     0 ——→ empty
               1 ——→ PRIORITY
```

```
Example: CALL METHOD;
            ↓
         10METHOD000;
Example: CALL ABC(A,B*C,D);
            ↓
         10ABC(AƀB,C,*ƀD)000;
Example: CALL WRITE (A,B) TASK PRIORITY (I+J);
            ↓
```

```
        10WRITE(AØB)101(I,J,+);
Example: CALL WRITE(A,B) PRIORITY (I+J) TASK (T4);
        10WRITE(AØB)1(T4)01(I,J,+);
```

2.4.4 The DECLARE (DCL) Statement and the DEFAULT Statement

The DECLARE statement and DEFAULT statement are non-executable statements. The routine DICT will scan them and build name tables which save some information for pass 2. If there is any error in them routine DICT will give the error message.

General Format:

$$\left\{\begin{array}{l}\text{DECLARE}\\ \text{DCL}\end{array}\right\}\text{etc.;}$$

Intermediate Language:

20;

General Format:

DEFAULT etc.;

Intermediate Language:

21;

```
Example: DECLARE 1 V LIKE W, 1 L, 2 N FLOAT;
                 20;
Example: DEFAULT RANGE (P:R) POINTER;
                 21;
```

2.4.5 The DISPLAY Statement

General Format:

DISPLAY(scalar-expression);

DISPLAY(scalar-expression)REPLY(scalar-character-variable)

[EVENT(scalar-event-variable)];

Intermediate Language:

type-code

70 (POLISH(scalar-expression));

type-code 1st field

70 (POLISH(scalar-expression)) 1(scalar-character-variable

2nd field

0

1 (scalar-event-variable) ;

1st field: 1 → REPLY

2nd field: 0 → empty

1 → EVENT

Example: DISPLAY('ERRORØINØEVALUATION');

70('ERRORØINØEVALUATION');

Example: DISPLAY(5*N-2) REPLY(X) EVENT (BES);

70(5,N,*,2,-)1(X)1(BES);

Example: DISPLAY(5*N-2) EVENT(BES) REPLY (X);

70(5,N,*,2,-)1(X)1(BES);

2.4.6 The DO Statement

General Format:

```
DO;

DO WHILE(scalar-expression);

DO {pseudo-variable} =specification[,specification]...;
   {variable       }
```

A specification has the following format:

```
expression1 [TO expression2 [BY expression3]] [WHILE(expression4)]
            [BY expression3 [TO expression2]]
```

Intermediate Language:

type-code

11;

type-code

```
   11 (POLISH(scalar-expression));

type-code
      { pseudo-variable }
   11 { variable        }   =SPECIFICATION[,SPECIFICATION]...;
```

A SPECIFICATION has the following format:

```
                          1st field                2nd field

                          0                        0
(POLISH(expression1))
                          1(POLISH(expression2))   1(POLISH(expression3))

                          3rd field

                          0

                          1(POLISH(expression4))
```

```
1st field:     0 ---> empty

               1 ---> TO

2nd field:     0 ---> empty

               1 ---> BY

3rd field:     0 ---> empty

               1 ---> WHILE
```

```
Example: LOOP: DO;

         11;

Example: DO WHILE (3*X-5<9);

         11(3,X,*,5,-,9,<);

Example: DO X=3,7 TO 11 BY .5,4*I,SQRT(J),A WHILE(B¬=C);

         11X=(3)000,(7)1(11)1(.5)0,(4,I,*)000,(SQRT(J))000,

         (A)001(B,C,¬=);

Example: DO COMPLEX(X,Y)=0 BY 1+1I WHILE (X<10);

         11COMPLEX(Xb̸Y)=(0)01(1,1I,+)1(X,10,< );

Example: DO INDEX=Z WHILE (A >B), 5 TO 10 WHILE (A=B),100;

         11INDEX=(Z)001(A,B,> ),(5)1(10)01(A,B,=),(100)000;
```

Example: L: DO I=1 TO 9, 11 TO 20;

11I=(1)1(9)00,(11)1(20)00;

2.4.7 The END Statement

General Format:

END [label];

Intermediate Language:

type-code

41[label];

Example: END;

41;

Example: END TEST;

41TEST;

2.4.8 The ENTRY Statement

General Format:

{entry-name:} ...ENTRY[(parameter[,parameter]...)]

[RETURNS(data-attribute-list)];

Intermediate Language:

type-code 1st field

43 [(parameter[ƀparameter]...)] 0 ;
1(DATA-ATTRIBUTE-LIST)

1st field: 0 ⟶ empty

1 ⟶ RETURNS

For the definition of DATA-ATTRIBUTE-LIST, see 2.4.20.

Example: L1:L2: ENTRY (X) RETURNS (FIXED BINARY);

43(X)1(03ƀ01);

Example: A:ENTRY (X,N,Y,M);

43(XƀNƀYƀM)0;

Example: B:ENTRY;

430;

2.4.9 The EXIT Statement

General Format:

EXIT;

Intermediate Language:

type-code

12 ;

Example: EXIT;

12;

2.4.10 The FORMAT Statement

General Format:

{label:}...FORMAT(format-list);

Intermediate Language:

type-code

78 (FORMAT-LIST);

For the definition of FORMAT-LIST, see 2.4.21.

Example: COMMON: FORMAT(A(5),F(5,2),X(3),F(10,0));

78(08(5),01(5b̸2),11(3),01(10b̸0));

2.4.11 The GET Statement

General Format:

GET option-list;

The format of option-list:

FILE(filename) [COPY] [SKIP[(scalar-expression)]]
STRING(scalar-character-string-expression)
BITSTRING(scalar-bit-string-expression)

[data-specification]

General format of data-specification:

LIST(data-list)

DATA[(data-list)]

EDIT{(data-list)(format-list)}...

Intermediate Language:

type-code	1st field	2nd field
	0	0
81	1(filename) 2(POLISH(scalar-character-string-expression)) 3(POLISH(scalar-bit-string-expression))	1

3rd field	4th field
0	0 1(DATA-LIST)
1[(POLISH(scalar-expression))]	2[(DATA-LIST)] ;
	3{(DATA-LIST)(FORMAT-LIST)}...

1st field: 0 → empty

1 → FILE

2 → STRING

3 → BITSTRING

2nd field: 0 → empty

1 → COPY

3rd field: 0 → empty

1 → SKIP

If 1st field is not 1(filename), then the only one correct combination of 2nd field and 3rd field is 00. Error message will be printed out for combinations of 10, 01, and 11. (See rules 10 and 11 in the GET Statement in IBM (1969).)

4th field: 0 → empty

1 → LIST

2 ⟶ DATA

3 ⟶ EDIT

For the definitions of FORMAT-LIST and DATA-LIST, see 2.4.21 and 2.4.22 respectively.

Example: GET LIST (A,B,C);

810001(Ab̸Bb̸C);

Example: GET FILE (BETA) EDIT (X,Y,Z) (A(5),F(5,2),A(10));

811(BETA)003(Xb̸Yb̸Z)(08(5),01(5b̸2),08(10));

Example: GET FILE(INPUT) SKIP(A||B) EDIT(L,K(3/C),T,((R(I,J) DO J=1 TO 6), S(I) DO I=1 TO 3)) (LINE (4*12),F (7,1), 2E(10,3),SKIP(3), (3) ((3*2)F(8,2),P'99999'));

811(INPUT)01(A,B,||)3(L;b̸K(3,C,/)b̸Tb̸((R(Ib̸J)11J= (1)1(6)00)b̸S(I)11I=(1)1(3)00))(17(4,12,*),01(7b̸1), (2)02(10b̸3),13(3),(3)((3,2,*)01(8b̸2),04'99999'));

<u>2.4.12 The GO TO (GOTO) Statement</u>

General Format:

$$\left\{\begin{array}{l}\text{GO TO}\\ \text{GOTO}\end{array}\right\}\left\{\begin{array}{l}\text{label-constant}\\ \text{scalar-label-variable}\end{array}\right\};$$

Intermediate Language:

type-code

$$13\left\{\begin{array}{l}\text{label-constant}\\ \underline{\text{scalar-label-variable}}\end{array}\right\};$$

Example: GO TO LABEL;

13LABEL;

Example: GOTO SWITCH(I);

13SWITCH(I);

Example: GOTO ABC (I+J,I-J,I*J);

```
13ABC(I,J,+ØI,J,-ØI,J,*);
```

2.4.13 The IF Statement

General Format:

```
IF scalar-expression THEN unit-1 [ELSE unit-2]
```

Intermediate Language:

```
type-code
   14     POLISH(scalar-expression)(part-1)[*(part-2)];
```

part-i is the corresponding intermediate language code for unit-i with the last ';' deleted. For example:

If unit-i is S(1); S(2); S(3); ..S(n-1); S(n);

then part-i will be I(1);I(2);I(3);..I(n-1);I(n)

Where I(j) is the intermediate language code of S(j).

```
Example: IF GREEN THEN DO I=0; GREEN='0'B;END;
         14GREEN(11I=(0)000;GREEN='0'B;41);
Example: IF LAST THEN GOTO ST6;ELSE X=9;
         14LAST(13ST6)*(X=9);
Example: IF X=0
              THEN IF Y=1
                     THEN X=1;
                     ELSE;
              ELSE  GO TO NEXT;
   14X,0,=(14Y,1,=(X=1)*(15))*(13NEXT);
Example: IF BA=Z THEN CALL X(0); ELSE CALL X(A);
         14BA,Z,=(10X(0)000)*(10X(A)000);
```

```
Example: IF X=0
                THEN IF Y=1
                       THEN X=1;
                ELSE  GO TO NEXT;
     14X,0,=(14Y,1,=(X=1)*(13NEXT));
Example: IF X>Y
                THEN IF Z=W
                   THEN L:  Y=1;
                   ELSE;
                ELSE (SIZE):  Y=A;
     14X,Y,>(14Z,W,=(Y=1)*(15))*((05):Y=A);
Example: IF X+Y THEN L:  BEGIN:
                          A=B*C;
                          IF A<0 THEN DO;
                                      B=1;
                                      C=0;
                                      END;
                          END;
                ELSE(SIZE):LL: IF B=1
                                      THEN CALL MAX(X,Y);
                                      ELSE CALL MIN(X,Y);
     14X,Y,+(4000;A=B,C,*;14A,0,<(11;B=1;C=0;41);41)*((05):
         14B,1,=(10MAX(Xb̸Y)000)*(10MIN(Xb̸Y)000));
```

2.4.14 The NULL Statement

General Format:

```
;
```

Intermediate Language:

type-code

15;

Example: LABEL:;
↓
15;

Example: ;
↓
15;

Example: (SIZE):L:;
↓
(05):15;

2.4.15 The PROCEDURE (PROC) Statement

General Format:

$\left\{\text{entry-name:}\right\}\ldots\left\{\begin{array}{l}\text{PROCEDURE}\\ \text{PROC}\end{array}\right\}$ [(parameter[,parameter]...)]

[RETURNS(data-attribute-list)] [OPTIONS(option-list)]

[ORDER|REORDER] [RECURSIVE];

Intermediate Language:

type-code		1st field
		0
42 [(parameter[ꝥparameter]...)]		
		1(DATA-ATTRIBUTE-LIST)
2nd field	**3rd field**	**4th field**
0	0	0
	1	;
1(option-list)	2	1

1st field: 0 → empty

1 → RETURNS

For the definition of DATA-ATTRIBUTE-LIST, see 2.4.20.

2nd field: 0 → empty

1 → OPTIONS

3rd field: 0 → empty

1 → ORDER

2 → REORDER

4th field: 0 → empty

1 → RECURSIVE

Example: A:I: PROCEDURE;

420000;

Example: A: PROC (B,C) RETURNS (FIXED);

42(Bb̸C)1(03)000;

Example: LAB: PROCEDURE (A,B,CC) RECURSIVE REORDER

OPTIONS (MAIN, SYSTEM) RETURNS (FLOAT REAL

DECIMAL (12,8));

42(Ab̸Bb̸CC)1(04b̸05b̸02b̸(12b̸8))1(MAINb̸SYSTEM)21;

<u>2.4.16 The PUT Statement</u>

General Format:

PUT option-list;

The format of option-list:

$$\left[\begin{array}{l}\text{FILE(filename)}\\ \text{STRING(scalar-character-string-variable)}\\ \text{BITSTRING(scalar-bit-string-variable)}\end{array}\right]$$ [PAGE] [SKIP[(expression)

[LINE(expression)] [data-specification]

General format of data-specification:

LIST(data-list)

DATA[(data-list)]

EDIT{(data-list)(format-list)}...

Intermediate Language:

type-code	1st field	2nd field
	0	0

1(filename)

71

2(<u>scalar-character-string-variable</u>)

3(<u>scalar-bit-string-variable</u>) 1

3rd field 4th field

0 0

1[(POLISH(expression))] 1(POLISH(expression))

5th field

0

1(DATA-LIST)

2[(DATA-LIST)] ;

3{(DATA-LIST)(FORMAT-LIST)}...

1st field: 0 → empty

1 → FILE

2 → STRING

3 → BITSTRING

2nd field: 0 → empty

1 → PAGE

3rd field: 0 → empty

1 → SKIP

4th field: 0 → empty

1 → LINE

If 1st field is 2(<u>scalar-character-string-variable</u>), then 2nd, 3rd, and 4th fields must be all 0. Error message will be printed out if any of them is 1.

```
5th field:      0 → empty
                1 → LIST
                2 → DATA
                3 → EDIT
```

For the definitions of FORMAT-LIST and DATA-LIST, see 2.4.21 and 2.4.22 respectively.

```
Example: PUT DATA (A,B,C);
         7100002(Ab̸Bb̸C);

Example: PUT FILE(LIST) EDIT(X,Y,Z)(A(10)) PAGE;
         711(LIST)1003(Xb̸Yb̸Z)(08(10));

Example: PUT FILE(OUT) EDIT(NAME,NO)(A(30),F(5))
         PAGE LINE (10);
         711(OUT)101(10)3(NAMEb̸NO)(08(30),01(5));
```

2.4.17 The RETURN Statement

General Format:

```
RETURN[(scalar-expression)];
```

Intermediate Language:

```
type-code
   16     [(POLISH(scalar-expression))];
```

```
Example: OUT: RETURN;
         16;

Example: RETURN(X**2+Y**2);
         16(X,2,**,Y,2,+);
```

2.4.18 The STOP Statement

General Format:

```
STOP;
```

Intermediate Language:

type-code

17;

Example: STOP;

17;

<u>2.4.19 The Other Statements</u>

General Format:

STATEMENT-IDENTIFIER etc.;

Intermediate Language:

type-code

99;

The other statements not defined in previous sections have not been implemented in the present verion. If any statement in this group is used in source program, the intermediate language is always 99; and an error message,

```
UMIP112I XXXXXX %%ERROR%% THE 'STATEMENT-IDENTIFIER' STATEMENT HAS
                    NOT BEEN IMPLEMENTED YET.  DELETES THE REST.
         XXXXXX (THIS ERROR (OR WARNING) MESSAGE IS ISSUED FOR
                    STATEMENT NUMBER xxx.)
```

, will be printed out immediately after that statement listing. xxx is a 3-digit statement number.

<u>2.4.20 DATA-ATTRIBUTE-LIST</u>

DATA-ATTRIBUTE-LIST has corresponding code for each attribute in data-attribute-list. The former is in the same order as the latter. The codes of DATA-ATTRIBUTE-LIST are separated by exactly one blank.

data-attribute	corresponding code
BINARY	01

DECIMAL	02
FIXED	03
FLOAT	04
REAL	05
COMPLEX	06
(precision)	(<u>precision</u>) i.e., (p,g) → (pbg)
	(p) → (p)

<u>precision</u> has same contents as precision has. The former has exactly one blank to replace the delimiter ',' of the latter.

Examples: (10,-2) → (10b-2)

(10,+1) → (10b+1)

(10,1) → (10b1)

(10) → (10)

PICTURE'picture-specification' → 07'picture-specification'

Example:

PICTURE '+9999E-99' → 07'+9999E-99'

BIT	08
CHARACTER	09
(length)	(length) i.e., (l) → (l)
	and l=* or constant only
VARYING	11
AREA	12
(size)	(size) i.e., (s) → (s)
	and s=* or constant only
ENTRY	13
EVENT	14
OFFSET	15

OFFSET(scalar-area-variable)	15(scalar-area-variable)
POINTER	16
TASK	17

2.4.21 FORMAT-LIST

If format-list is: item[,item]...

then FORMAT-LIST is: ITEM[,ITEM]...

Where 'item' may be: format-item

n format-item

n (format-list)

so the definition of 'item' is recursive.

If item=n(format-list) then ITEM=N(FORMAT-LIST)

If item=n format-item then ITEM=N FORMAT-ITEM

If item= format-item then ITEM= FORMAT-ITEM

The definition of 'ITEM' is recursive too!

If n=(expression) then N=(POLISH(expression))

If n=constant then N=(constant)

Let X=POLISH(x)

Define a mapping MAP such that

FORMAT-ITEM=MAP(format-item)

by the following correspondence:

format-item	MAP(format-item)
F(w)	01(W)
F(w,d)	01(Wb̸D)
F(w,d,p)	01(Wb̸Db̸P)
E(w,d)	02(Wb̸D)
E(w,d,s)	02(Wb̸Db̸S)
C(r)	03(MAP(r))
C(r,s)	03(MAP(r),MAP(s))

Where r,s ε {F format-item, E format-item, P format-item}

Example: let format-item=C(F(I+J,I-J),E(10,3))

then MAP(format-item)=MAP(C(F(I+J,I-J),E(10,3)))

=03(MAP(F(I+J,I-J)),MAP(E(10,3)))

=03(01(I,J,+b̸I,J,-),02(10b̸3))

P'numeric-picture-specification'

04'numeric-picture-specification'

P'character-string-picture-specification'

04'character-string-picture-specification'

BP'binary-numeric-picture-specification'

05'binary-numeric-picture-specification'

format-item	MAP(format-item)
B(w)	06(W)
B	06
BB(w)	07(W)
BB	07
A(w)	08(W)
A	08
X(w)	11(W)

BX(w)	12(W)
SKIP(w)	13(W)
SKIP	13
COLUMN(w)	14(W)
BCOLUMN(w)	15(W)
PAGE	16
LINE(w)	17(W)
R(statement-label-designator)	18(statement-label-designator)

Where statement-label-designator is either a label constant or a label variable.

2.4.22 DATA-LIST

If data-list is: element[,element]...

then DATA-LIST is: ELEMENT[ȸELEMENT]...

Where ELEMENT=POLISH(element) if the element is an expression.

ELEMENT=(ELEMENT[ȸELEMENT]...*) if the element is a repetitive specification, i.e.,

$$\text{(element[,element]...DO}\left\{\begin{array}{l}\text{scalar-variable}\\ \text{scalar-pseudo-variable}\end{array}\right\}=$$

specification[,specification]...)

Where a specification has the following format:

$$\text{expression1}\left[\begin{array}{l}\text{TO expression2 [BY expression3]}\\ \text{BY expression3 [TO expression2]}\end{array}\right]\text{[WHILE(expression4)]}$$

and * stands for the intermediate language code of the third form of the DO statement.

The definition of the repetitive specification is recursive. It may be nested to any depth.

3. STATEMENT RECOGNIZER

The purpose of this chapter is to present a brief description of the techniques and principles used in the compiler program. The compiler program contains five procedures and has more than 7300 cards. External procedure STATEMENT.RECOGNIZER has thirty-seven internal procedures. Procedure POLISH is the largest procedure in procedure STATEMENT.RECOGNIZER. It has more than 1100 cards. Procedure STATEMENT.RECOGNIZER is composed of two groups of procedures. The first group is the Generator Group and the second group is the Utility Group. Figures 2, 3, 4, and 5 give a rough structure of the compiler program.

```
[ST.REKNZER.C:

[STATEMENT.RECOGNIZER:

[ERROR:

[READ:

[DICT:
```

Figure 2. Structure of the compiler program.

Generator Group

Utility Group

Figure 3. Structure of the procedure STATEMENT.RECOGNIZER.

ASSIGN:

BEGIN:

CALL:

DECLARE:

DEFAULT:

DISPLAY:

DO:

END:

ENTRY:

EXIT:

FORMAT:

GET:

GO:

IF:

NULL:

PROCEDURE:

PUT:

RETURN:

STOP:

CLOSE:DELAY:...etc.

ELSE:

Figure 4. Structure of the Generator Group.

READB:

DICTB:

VARIBL:

DLIST2:

DLIST3:

ATRIBU:

ASSIGNMENT:

POLISH:

ALPHAOPT:

INTEGER?:

HANDLEASSIGN:

LEGAL?ID:

FOMAT:

COND.PREFIX:

POLI:

DEBUG:

Figure 5. Structure of the Utility Group.

3.1 Expression Translator

Procedure POLISH is the most important procedure among the procedures contained in the external procedure STATEMENT.RECOGNIZER. It is a recursive procedure. (See 3.2.) Its function is to transform an infix expression into Reverse Polish expression.

3.1.1 Definition of Reverse Polish Expression

The definition of the Reverse Polish expression for the proposed PL/I compiler is given below.

RPexpression ← operand,operand,operator | operand

operand ⟵ RPexpression | operand,¬ | operand,NOT | operand,-P

Where the vertical stroke | indicates that a choice is to be made and -P stands for prefix - (prefix minus). Prefix + will be ignored by the compiler. An operator is any operator defined in the PL/I operator set. It may be an alphabetic operator. An operand is a variable or a constant. If it is a subscripted variable then the subscripts in it must have been converted into RPexpressions.

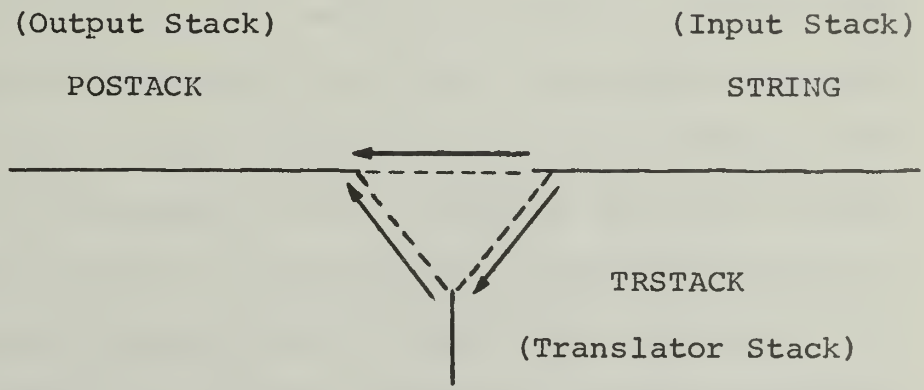
Example: 12+x-y/z*9 is translated into Reverse Polish expression

12,x,+,y,z,/,9,*,-

It can be seen that operands in the Reverse Polish expression appear in the same order as in the original infix expression but the operators have been reordered according to the priorities of the operators.

3.1.2 Conversion of Infix Expression

Dijkstra (1961) shows the conversion process has many analogies with a three way railraod shunting yard of the following form.



He uses these analogies to explain how an infix expression can be converted into a Reverse Polish expression by comparing the priority numbers of two delimiters. A set of priority numbers

is given below.

Delimiter	Priority Number
¬ ,**,Prefix +, Prefix -	8
*,/	7
Infix +, Infix -	6
>=,> , ¬>,¬ =, < ,¬<, <=,=	5
&	4
|	3
||,)	2
(	1

Where prefix + will be neglected by the compiler and prefix - will be replaced by the symbol '-P'. Alphabetic operators have the same priority numbers as the corresponding symbol operators. The rule is that an incoming operand passes straight from STRING stack to POSTACK stack. The incoming operator is pushed into the top of the TRSTACK stack and then its corresponding priority number is pushed into the top of the TRSTACK stack, but before the delimiter is shunted into the TRSTACK stack, operators with their priority numbers discarded and in the top of the TRSTACK stack are transported from it to the POSTACK stack as long as their priority numbers are greater than or equal to the priority number of the new operator. If an expression involves operations of the same priority, the operations ¬, **, prefix +, and prefix - are performed from right to left and the other operations are performed from left to right. Thus, if the priority number of the new operator is equal to 8 and is equal to the priority number of the operator which is in the second level of the TRSTACK

stack, no unstacking will be done. The delimiter '(' must be stacked in the top of the TRSTACK stack without first doing any unstacking. This is because all of the following input up to the matching ')' has a higher priority than anything currently in the TRSTACK stack. The lowest priority number must be given to the delimiter '(' and a second lowest priority number must be given to the delimiter ')' so that the TRSTACK stack will be emptied down to but not including the left parenthesis when the right parenthesis is being sensed. Therefore, the top of the TRSTACK stack may be examined after the automatic unstacking to check if the left parenthesis is indeed there. If it is there, it will be discarded. The right matching parenthesis is never stacked into the TRSTACK stack. The operands and operators are separated by commas.

Example: 12+x-y/z*9

is translated into

12,x,+,y,z,/,9,*,-

STRING: 12+x-y/z*9

TRSTACK:

POSTACK:

STRING: +x-y/z*9

TRSTACK:

POSTACK: 12

STRING: x-y/z*9

TRSTACK: 6,+ Since TRSTACK was empty, + and 6 should

POSTACK: 12 be stacked into TRSTACK stack

STRING: -y/z*9

TRSTACK: 6,+

POSTACK: 12,x

STRING: y/z*9

TRSTACK: 6,-

POSTACK: 12,x,+

STRING: /z*9

TRSTACK: 6,-

POSTACK: 12,x,+,y

STRING: z*9

TRSTACK: 7,/,6,-

POSTACK: 12,x,+,y

STRING: *9

TRSTACK: 7,/,6,-

POSTACK: 12,x,+,y,z

STRING: 9

TRSTACK: 7,*,6,-

POSTACK: 12,x,+,y,z,/

STRING:

TRSTACK: 7,*,6,-

POSTACK: 12,x,+,y,z,/,9

STRING:

TRSTACK: 6,-

POSTACK: 12,x,+,y,z,/,9,*

STRING:

TRSTACK:

POSTACK: 12,x,+,y,z,/,9,*,-

Since there is no more input left in the STRING stack, the remaining items in the TRSTACK stack must be emptied.

Example: (u||v)-w+13*(x|y)-m*(-n/z**12*(x+13*m)+y)

is translated into

u,v,||,w,-,13,x,y,|,*,+,m,n,-P,z,

12,**,/,x,13,m,*,+,*,y,+,*,-

STRING: (u||v)-w+13*(x y)-m*(-n/z**12*(x+13*m)+y)

TRSTACK:

POSTACK:

STRING: (u||v)-w+13*(x y)-m*(-n/z**12*(x+13*m)+y)

TRSTACK: 1,(

POSTACK:

STRING: ||v)-w+13*(x|y)-m*(-n/z**12*(x+13*m)+y)

TRSTACK: 1,(

POSTACK: u

STRING: v)-w+13*(x|y)-m*(-n/z**12*(x+13*m)+y)

TRSTACK: 2,||,1,(

POSTACK: u

STRING:)-w+13*(x|y)-m*(-n/z**12*(x+13*m)+y)

TRSTACK: 2,||,1,(

POSTACK: u,v

STRING: -w+13*(x|y)-m*(-n/z**12*(x+13*m)+y)

TRSTACK:

POSTACK: u,v,||

STRING: w+13*(x|y)-m*(-n/z**12*(x+13*m)+y)

TRSTACK: 6,-

POSTACK: u,v,||

STRING: +13*(x|y)-m*(-n/z**12*(x+13*m)+y)

TRSTACK: 6,-

POSTACK: u,v,||,w

STRING: 13*(x|y)-m*(-n/z**12*(x+13*m)+y)

TRSTACK: 6,+

POSTACK: u,v,||,w,-

```
STRING:   *(x|y)-m*(-n/z**12*(x+13*m)+y)

TRSTACK: 6,+

POSTACK: u,v, ||,w,-,13


STRING:   (x|y)-m*(-n/z**12*(x+13*m)+y)

TRSTACK: 7,*,6,+

POSTACK: u,v, ||,w,-,13


STRING:   x|y)-m*(-n/z**12*(x+13*m)+y)

TRSTACK: 1,(,7,*,6,+

POSTACK: u,v,||,w,-,13


STRING:   y)-m*(-n/z**12*(x+13*m)+y)

TRSTACK: 3,|,1,(,7,*,6,+

POSTACK: u,v, ||,w,-,13,x


STRING:   -m*(-n/z**12*(x+13*m)+y)

TRSTACK: 7,*,6,+

POSTACK: u,v, ||,w,-,13,x,y,|


STRING:   m*(-n/z**12*(x+13*m)+y)

TRSTACK: 6,-

POSTACK: u,v, ||,w,-,13,x,y,| ,*,+


STRING:   (-n/z**12*(x+13*m)+y)

TRSTACK: 7,*,6,-

POSTACK: u,v, ||,w,-,13,x,y,| ,*,+,m
```

```
STRING:   -n/z**12*(x+13*m)+y)
TRSTACK:  1,(,7,*,6,-
POSTACK:  u,v, ||,w,-,13,x,y,| ,*,+,m

STRING:   n/z**12*(x+13*m)+y)
TRSTACK:  8,-P,1,(,7,*,6,-
POSTACK:  u,v, ||,w,-,13,x,y,| ,*,+,m

STRING:   z**12*(x+13*m)+y)
TRSTACK:  7,/,1,(,7,*,6,-
POSTACK:  u,v, ||,w,-,13,x,y,| ,*,+,m,n,-P

STRING:   12*(x+13*m)+y)
TRSTACK:  8,**,7,/,1,(,7,*,6,-
POSTACK:  u,v, ||,w,-,13,x,y,| ,*,+,m,n,-P,z

STRING:   (x+13*m)+y)
TRSTACK:  7,*,1,(,7,*,6,-
POSTACK:  u,v, ||,w,-,13,x,y,| ,*,+,m,n,-P,z,12,**,/

STRING:   x+13*m)+y)
TRSTACK:  1,(,7,*,1,(,7,*,6,-
POSTACK:  u,v, ||,w,-,13,x,y,| ,*,+,m,n,-P,z,12,**,/

STRING:   13*m)+y)
TRSTACK:  6,+,1,(,7,*,1,(,7,*,6,-
POSTACK:  u,v, ||,w,-,13,x,y,| ,*,+,m,n,-P,z,12,**,/,x
```

STRING: m)+y)

TRSTACK: 7,*,6,+,1,(,7,*,1,(,7,*,6,-

POSTACK: u,v,||,w,-,13,x,y,|,*,+,m,n,-P,z,12,**,/,x,13

STRING: +y)

TRSTACK: 7,*,1,(,7,*,6,-

POSTACK: u,v,||,w,-,13,x,y,|,*,+,m,n,-P,z,12,**,/,x,13,m,*,+

STRING: y)

TRSTACK: 6,+,1,(,7,*,6,-

POSTACK: u,v,||,w,-,13,x,y,|,*,+,m,n,-P,z,12,**,/,x,13,m,*,+,*

STRING:

TRSTACK: 7,*,6,-

POSTACK: u,v,||,w,-,13,x,y,|,*,+,m,n,-P,z,12,**,/,x,13,m,*,+,*,y,+

STRING:

TRSTACK:

POSTACK: u,v,||,w,-,13,x,y,|,*,+,m,n,-P,z,
12,**,/,x,13,m,*,+,*,y,+,*,-

3.2 Recursive Procedures

A recursive function is a function which is defined in terms of itself. The factorial function for non-negative integral argument is possibly the best known example of a recursively defined function. It has the following definition:

(n)!=n*(n-1)! if n>0

(0)!=1

Any recursive definition must have an explicit definition for some value or values of the argument(s), otherwise the definition would be circular. The idea of a procedure which itself calls another procedure is very familiar; the analogue of a recursive function in these terms is a procedure which contains call(s) to itself. This kind of procedure is called recursive procedure. There are several recursive procedures in the statement recognizer. One of them is procedure POLISH.

The examples presented in the previous section contained no subscripted variable. To include this kind of variable, a special delimiter, say S, must be attached to the set of priority numbers given in 3.1.2. S has the lowest priority number 0. It is used as an indicator to mark the beginning of an expression, therefore S and 0 must be stacked <u>once</u> into the top of the TRSTACK stack whenever a <u>new</u> expression is going to be recognized. In general, the compiler considers it is at the end of an expression whenever an empty STRING stack or an extra comma is being sensed. A semicolon in the top of the PARTCODE stack is used as a signal to make the compiler consider it is also at the end of an expression whenever an extra right parenthesis is being sensed. It is the case that procedure POLISH is called to recognize a subscript in a subscripted variable. The TRSTACK stack will be emptied down to but not including the first (0,S) pair when the end of that expression is encountered. The (0,S) pair will be deleted then. The subscripts separator (i.e., comma) in the subscripted variable will be replaced by exactly one blank and the subscripts will be expressed in Reverse Polish expressions.

Example: 10+x(m+n,m-n)*y-z

is translated into

10,x(m,n,+ꞓm,n,-),y,*,+,z,-

Where ꞓ stands for one blank.

STRING: 10+x(m+n,m-n)*y-z

TRSTACK:

POSTACK:

STRING: 10+x(m+n,m-n)*y-z

TRSTACK: 0,S

POSTACK:

STRING: +x(m+n,m-n)*y-z

TRSTACK: 0,S

POSTACK: 10

STRING: x(m+n,m-n)*y-z

TRSTACK: 6,+,0,S

POSTACK: 10

STRING: (m+n,m-n)*y-z

TRSTACK: 6,+,0,S

POSTACK: 10,x

STRING: m+n,m-n)*y-z

TRSTACK: 6,+,0,S

POSTACK: 10,x('(' is part of the <u>subscripted variable</u>.

STRING: m+n,m-n)*y-z

TRSTACK: 0,S,6,+,0,S

POSTACK: 10,x(

STRING: +n,m-n)*y-z

TRSTACK: 0,S,6,+,0,S

POSTACK: 10,x(m

STRING: n,m-n)*y-z

TRSTACK: 6,+,0,S,6,+,0,S

POSTACK: 10, x(m

STRING: ,m-n)*y-z

TRSTACK: 6,+,0,S,6,+,0,S

POSTACK: 10,x(m,n

STRING: ,m-n)*y-z

TRSTACK: 0,S,6,+,0,S

POSTACK: 10,x(m,n,+

STRING: ,m-n)*y-z

TRSTACK: 6,+,0,S

POSTACK: 10,x(m,n,+

STRING: m-n)*y-z

TRSTACK: 6,+,0,S

POSTACK: 10,x(m,n,+ø

STRING: m-n)*y-z

TRSTACK: 0,S,6,+,0,S

POSTACK: 10,x(m,n,+b̸

STRING: -n)*y-z

TRSTACK: 0,S,6,+,0,S

POSTACK: 10,x(m,n,+b̸m

STRING: n)*y-z

TRSTACK: 6,-,0,S,6,+,0,S

POSTACK: 10,x(m,n,+b̸m

STRING:)*y-z

TRSTACK: 6,-,0,S,6,+,0,S

POSTACK: 10,x(m,n,+b̸m,n

STRING:)*y-z

TRSTACK: 0,S,6,+,0,S

POSTACK: 10,x(m,n,+b̸m,n,-

STRING:)*y-z

TRSTACK: 6,+,0,S

POSTACK: 10,x(m,n,+b̸m,n,-

STRING: *y-z

TRSTACK: 6,+,0,S

POSTACK: 10,x(m,n,+b̸m,n,-)

```
STRING:   y-z
TRSTACK: 7,*,6,+,0,S
POSTACK: 10,x(m,n,+bm,n,-)

STRING:   -z
TRSTACK: 7,*,6,+,0,S
POSTACK: 10,x(m,n,+bm,n,-),y

STRING:   z
TRSTACK: 6,-,0,S
POSTACK: 10,x(m,n,+bm,n,-),y,*,+

STRING:
TRSTACK: 6,-,0,S
POSTACK: 10,x(m,n,+bm,n,-),y,*,+,z

STRING:
TRSTACK: 0,S
POSTACK: 10,x(m,n,+bm,n,-),y,*,+,z,-

STRING:
TRSTACK:
POSTACK: 10,x(m,n,+bm,n,-),y,*,+,z,-
```

By the definition of the IF statement, procedure IF will call itself if it contains an IF statement in either unit-1 or unit-2 or both. (See 2.4.13 and 3.4.1.14.)

Procedure FOMAT is a procedure which will recognize a format-list for edit-directed data specification. Because format-list is recursively defined, procedure FOMAT was so coded that it is also a recursive procedure. While procedure FOMAT is recognizing an item which is in a format-list and has the form of n(format-list), it has to call itself. (See 2.4.21.)

Procedures DLIST2 and DLIST3 recognize a data-list. An element in a repetitive specification which appears in a data-list may be a repetitive specification again. Although the definition of the repetitive specification which may be an element of a data-list is recursive, procedures DLIST2 and DLIST3 are not recursive. Both procedures employ the stack TEMPA to count the depth of nest and get rid of the recursive problems. (See 2.4.22.)

3.3 External Procedures

There are five external procedures: ST.REKNZER.C, STATEMENT.RECOGNIZER, ERROR, READ, and DICT.

3.3.1 Procedure ST.REKNZER.C

This is the MAIN procedure and is not considered a part of the compiler program. The function of this procedure is to initiate the procedure STATEMENT.RECOGNIZER and then to print out the intermediate language code which was stored in file FFILE by procedure STATEMENT.RECOGNIZER.

3.3.2 Procedure STATEMENT.RECOGNIZER

The interior of the procedure STATEMENT.RECOGNIZER is the framework of the compiler program. Figure 6 is a rough flow diagram of the procedure STATEMENT.RECOGNIZER.

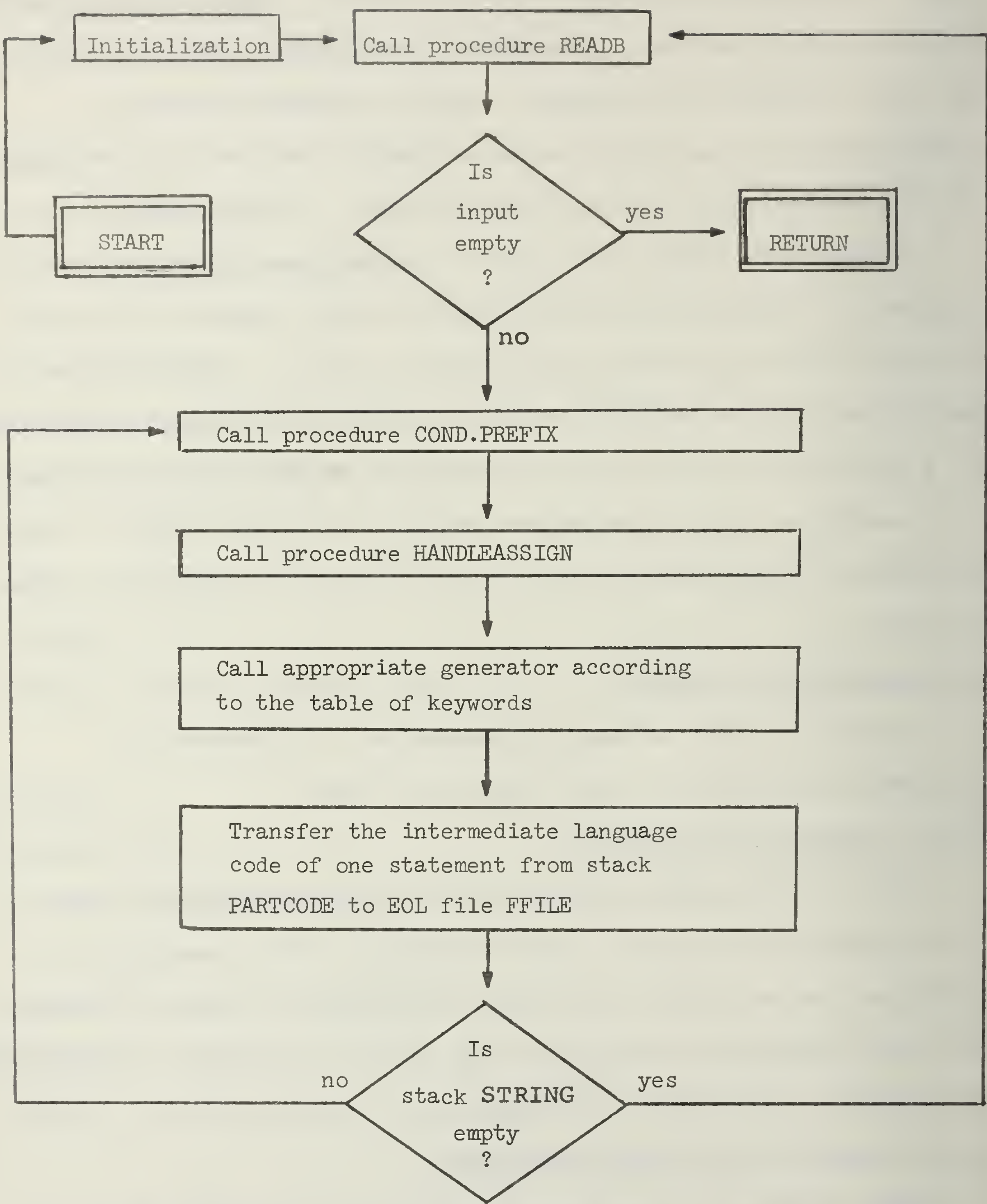


Figure 6. Flow diagram of the procedure STATEMENT.RECOGNIZER

3.3.3 Procedure ERROR

When an error is found procedure STATEMENT.RECOGNIZER calls the procedure ERROR to print out an error message. After this, the compiler continues to analyze the same statement as long as possible. However, the succeeding errors may occur simply due to the compiler's action to the preceding error(s). It was intended to make the error messages self explanatory. A message headed by %WARNING% implies the statement is still okay, if the programmer agrees with the compiler's action. A message headed by %%ERROR%% implies the statement is not okay at all and the programmer has to modify it. The information needed by the procedure ERROR is stored in stack TEMP. Procedure ERROR first prints out the error title number, then goes to the corresponding section for printing the appropriate error message, then prints out the number of that statement in which this error was found and finally returns to the calling procedure. Procedure ERROR usually tells the programmer where the error was found and sometimes prints page heading and number.

Example:

```
131            GO NEWYORK;
UMIP153I XXXXXX %WARNING% 'TO' INSERTED BEFORE 'NEWYORK'
                IN GO TO STATEMENT.
         XXXXXX (THIS ERROR (OR WARNING) MESSAGE IS ISSUED FOR
                STATEMENT NUMBER.131.)
```

Example:

```
293            Y=A-*B+C;
UMIP122I XXXXXX %%ERROR%% MISSING AN OPERAND BEFORE '*B+C'.
```

POSSIBLY ILLEGAL COMBINATION

XXXXXX OF OPERATORS EXISTS IN THE EXPRESSION.

XXXXXX (THIS ERROR (OR WARNING) MESSAGE IS ISSUED FOR

STATEMENT NUMBER 293.)

3.3.4 Procedure READ

When procedure STATEMENT.RECOGNIZER calls procedure READ, procedure READ prints page heading and number if necessary, deletes last eight columns of the input card, lists the current statement number and the contents of that input card, deletes the first column of that input card and returns one statement with a specified format in stack STRING to the calling procedure.

3.3.5 Procedure DICT

The function of the procedure DICT is to build up name tables which save information needed in pass 2. Every time an identifier is encountered, procedure STATEMENT.RECOGNIZER calls procedure DICT. (See Appendix B.)

3.4 Internal Procedures

External procedure STATEMENT.RECOGNIZER contains thirty-seven internal procedures which may be roughly classified into two logical groups: Generator Group and Utility Group. A brief description of each procedure will be given below.

3.4.1 Generator Group

This group has twenty-one procedures.

3.4.1.1 Procedure ASSIGN

Previous to compiler's call to this procedure, the compiler had recognized that the assignment statement had a correct left hand side. Thus the function of the procedure ASSIGN is to call

procedure POLISH and to check if this expression has a legal BY NAME option etc.

3.4.1.2 Procedure BEGIN

This procedure generates type-code 40 for the BEGIN statement. It uses stack F1 to hold information about ORDER or REORDER and uses stack F2 to hold information about OPTIONS. Before returning to the calling procedure, it concatenates stacks F1, F2 and semicolon to the end of the stack PARTCODE.

3.4.1.3 Procedure CALL

This procedure generates type-code 10 for the CALL statement. It uses stack F1 to hold information about option TASK, uses stack F2 to hold information about option EVENT and uses F3 to hold information about option PRIORITY. Before returning to the calling procedure, it concatenates stacks F1, F2, F3 and semicolon to the end of the stack PARTCODE.

3.4.1.4 Procedure DECLARE (DCL)

This procedure generates type-code 20 for the DECLARE statement.

3.4.1.5 Procedure DEFAULT

This procedure generates type-code 21 for the DEFAULT statement.

3.4.1.6 Procedure DISPLAY

This procedure generates type-code 70 for the DISPLAY statement. For the second form of the DISPLAY statement, it uses stack F1 to hold information about option REPLY and uses stack F2 to hold information about option EVENT. When it is ready to return to the calling procedure, it concatenates stacks

F1, F2 and semicolon to the end of the stack PARTCODE.

3.4.1.7 Procedure DO

The DO statement has three different formats. In general, they serve as PL/I statements. However, the third format without semicolon may be the partial contents of a repetitive specification in a data-list. Procedure DO will play different roles to recognize these two cases mentioned above. This purpose can be achieved by using a signal which will tell the procedure DO where the call to the procedure DO came from. If a semicolon is found in the top of the stack PARTCODE then procedure DO knows that this call came from procedure DLIST2 or DLIST3 which is used to recognize a data-list. If no semicolon is found then procedure DO will know that it has to recognize a general DO statement.

Because the information about option LIST or DATA or EDIT is stored in stack F5, the intermediate language code for the DO statement or for the partial contents of that repetitive specification will be stored in stack F5. Procedure DO attaches type-code 11 and some other information to the end of the stack F5. It uses stack F2 to hold inforamtion about TO, uses stack F3 to hold inforamtion about BY and uses stack F4 to hold information about WHILE. It then concatenates stacks F2, F3 and F4 to the end of the stack F5. Finally, if semicolon is found in the top of the stack PARTCODE then procedure DO will simply return to the calling procedure, otherwise the contents of the stack F5 will be transported to the stack PARTCODE, a semicolon will then be added to the end of that stack and then

procedure DO will return to the calling procedure.

<u>3.4.1.8 Procedure END</u>

This procedure generates type-code 41 for the END statement. If there is any label, it will follow the type-code.

<u>3.4.1.9 Procedure ENTRY</u>

This procedure generates type-code 43 for the ENTRY statement. It uses stack F1 to hold information about option RETURNS. It concatenates stack F1 and semicolon to the end of the stack PARTCODE before returning to the calling procedure.

<u>3.4.1.10 Procedure EXIT</u>

This procedure generates type-code 12 for the EXIT statement.

<u>3.4.1.11 Procedure FORMAT</u>

This procedure generates type-code 78 for the FORMAT statement and calls the procedure FOMAT to handle the format-list.

<u>3.4.1.12 Procedure GET</u>

This procedure generates type-code 81 for the GET statement. It uses stack F1 to hold information about FILE or STRING or BITSTRING, uses stack F2 to hold information about COPY, uses stack F3 to hold information about SKIP and uses stack F5 to hold information about LIST or DATA or EDIT. Procedures DLIST3 and/or FOMAT may be called if necessary. Before returning to the calling procedure, it concatenates stacks F1, F2, F3, F5 and semicolon to the end of the stack PARTCODE.

3.4.1.13 Procedure GO

This procedure generates type-code 13 for the GO TO statement. If there is any label, it will follow the type-code. It has an entry name called GOTO.

3.4.1.14 Procedure IF

The structure of the IF statement is very complicated because it may contain other statments, DO-groups or begin blocks. Procedure IF is recursive and has two internal procedures: procedure DOBEGIN handles DO-group and begin block, procedure HLABCON handles labels and condition prefixes. Figure 7 is a rough flow diagram of the procedure IF and Figure 8 is a rough flow diagram of the procedure DOBEGIN.

3.4.1.15 Procedure NULL

This procedure generates type-code 15 for the NULL statement.

3.4.1.16 Procedure PROCEDURE

This procedure has an entry name called PROC. It generates type-code 42 for the PROCEDURE statement. It uses stack F1 to hold information about option RETURNS, uses stack F2 to hold information about option OPTIONS, uses stack F3 to hold information about ORDER or REORDER and uses stack F4 to hold information about RECURSIVE. It concatenates stacks F1, F2, F3, F4 and semicolon to the end of the stack PARTCODE before returning to the calling procedure.

3.4.1.17 Procedure PUT

This procedure generates type-code 71 for the PUT statement. It uses stack F1 to hold information about FILE or STRING or BITSTRING, uses stack F2 to hold information about PAGE, uses

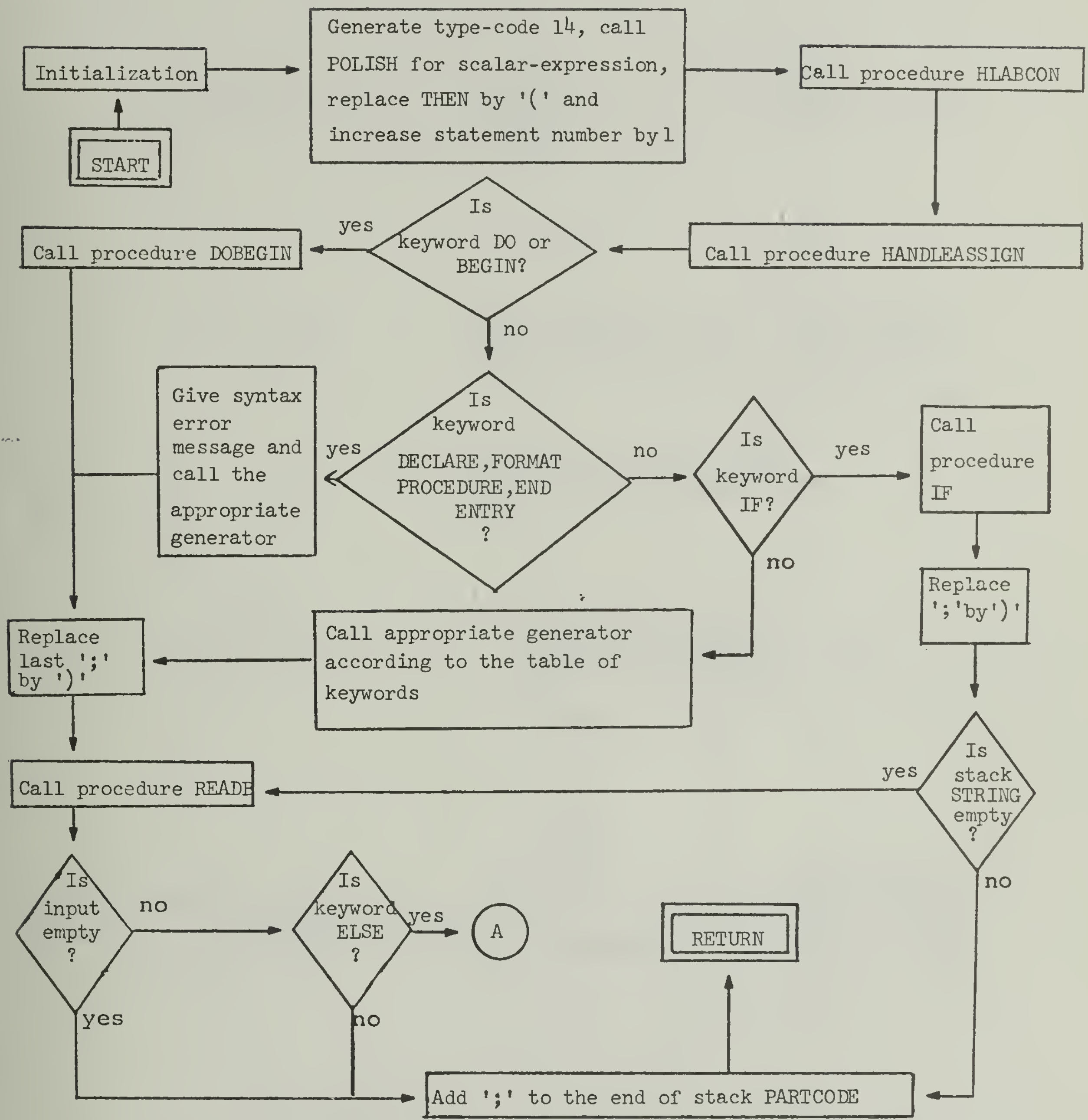


Figure 7. Flow diagram of the procedure IF.

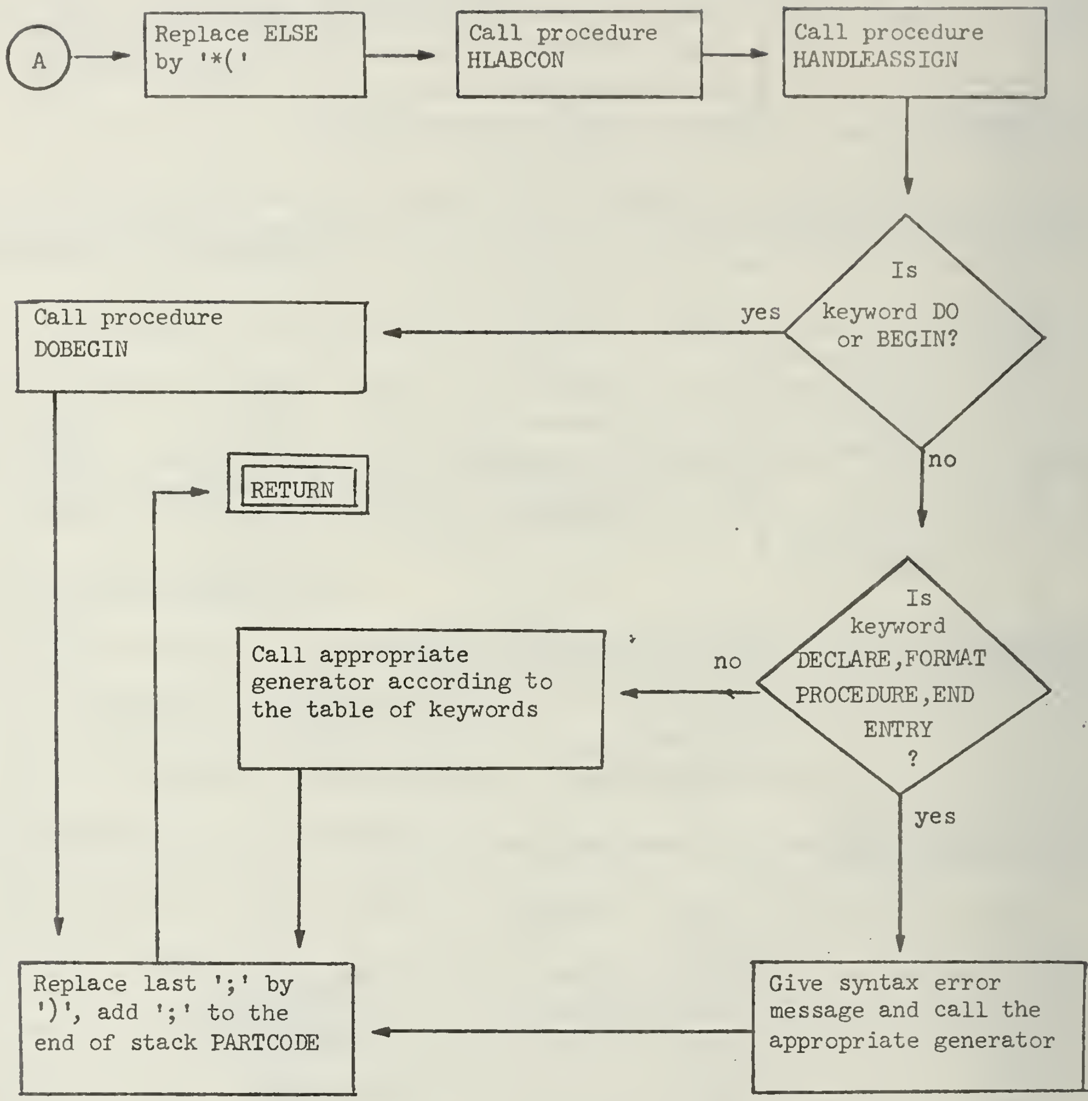


Figure 7. (Continued) Flow diagram of the procedure IF.

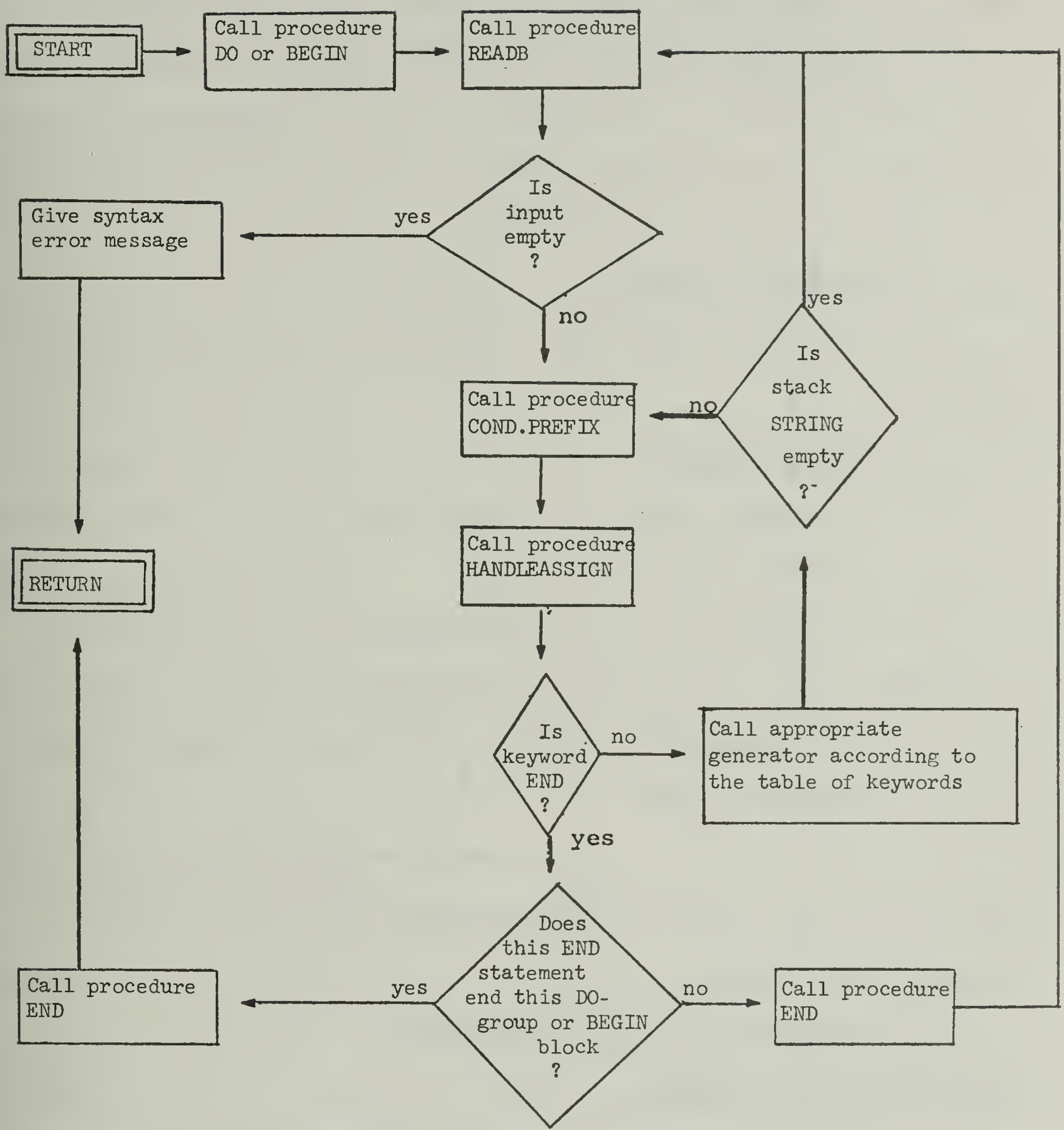


Figure 8. Flow diagram of the procedure DOBEGIN

stack F3 to hold information about SKIP, uses stack F4 to hold information about LINE and uses stack F5 to hold information about LIST or DATA or EDIT. Procedure DLIST3 or DLIST2 and/or FOMAT may be called if necessary. Before returning to the calling procedure, it concatenates stacks F1, F2, F3, F4, F5 and semicolon to the end of the stack PARTCODE.

3.4.1.18 Procedure RETURN

This procedure generates type-code 16 for the RETURN statement.

3.4.1.19 Procedure STOP

This procedure generates type-code 17 for the STOP statement.

3.4.1.20 Procedure CLOSE and So On

This procedure gives error message UMIP112I for those statements which have not been implemented in the present version such as statements CLOSE, DELAY, DELETE etc. (See 2.4.19.)

3.4.1.21 Procedure ELSE

This procedure gives error message UMIP183I for unmatched ELSE clause or for ELSE clause having a wrong IF clause.

3.4.2 Utility Group

This group has sixteen procedures.

3.4.2.1 Procedure READB

This procedure calls procedure READ and possibly calls procedure DEBUG according to the contents of the first card of the input data deck.

3.4.2.2 Procedure DICTB

This procedure calls procedure DICT and possibly calls procedure DEBUG according to the contents of the first card of the input data deck.

3.4.2.3 Procedure VARIBL

This procedure recognizes a single variable. If it is called in procedure DLIST3 then a semicolon, as a signal, will be found in the top of F5 stack. This makes compiler consider that star (*) subscripts in a subscripted variable are legal. If no semicolon is found in the top of F5 stack then star subscripts in a subscripted variable are not legal.

3.4.2.4 Procedure DLIST2

On output, this procedure recognizes a data-list for edit-directed and list-directed data specifications. It possibly sets a semicolon in the top of the stack PARTCODE and then calls procedure DO. (See 2.4.22.)

3.4.2.5 Procedure DLIST3

On input, this procedure recognizes a data-list for edit-directed and list-directed data specifications. On output, it recognizes a data-list for data-directed data specification. It possibly sets a semicolon in the top of the stack PARTCODE and then calls procedure DO. (See 2.4.22.)

3.4.2.6 Procedure ATRIBU

This procedure recognizes a data-attribute-list for procedures PROCEDURE and ENTRY.

3.4.2.7 Procedure ASSIGNMENT

This procedure analyzes a statement which is either an illegal assignment statement or a statement having undefined statement identifier. It assumes that the statement which is being scanned is an assignment statement and gives error messages accordingly.

3.4.2.8 Procedure POLISH

This procedure translates infix expression into Reverse Polish expression. (See 3.1 and 3.2.)

It has an internal procedure WTOGO which decides where to go when an error occurs.

3.4.2.9 Procedure ALPHAOPT

This procedure distinguishes whether an identifier is an alphabetic operator or not.

3.4.2.10 Procedure INTEGER?

This procedure recognizes an integer or a star (*). Its entry name INTEGER recognizes an integer only.

3.4.2.11 Procedure HANDLEASSIGN

This procedure recognizes the left hand side of an assignment statement. If a statement has a leading pattern which is roughly analogous to X,Y(I+J,5,K),Z(2),W=... or X=..., the keyword 'ASSIGN' will be pushed into the top of the stack STRING and then control calls procedure ASSIGN. If procedure HANDLEASSIGN cannot find a correct leading pattern, it will restore the original contents of the stack STRING into the stack STRING and then control calls the appropriate generator according to the keyword which is in the top level of the stack STRING etc.

3.4.2.12 Procedure LEGAL?ID

This procedure recognizes a legal identifier.

3.4.2.13 Procedure FOMAT

This procedure recognizes a format-list for edit-directed data specification.

3.4.2.14 Procedure COND.PREFIX

This procedure handles condition prefix(es).

3.4.2.15 Procedure POLI

This procedure calls procedure POLISH and will give an error message UMIP151I if a single star (*) appears in the top of the stack POSTACK.

3.4.2.16 Procedure DEBUG

This procedure prints out the contents of the stack STRING word by word. It may be called by procedures READB and/or DICTB. (See 3.4.2.1, 3.4.2.2, and Chapter 4.)

4. CONCLUSIONS AND SUGGESTIONS

The intermediate language described in Chapter 2 has proved to be a satisfactory design for the proposed PL/I compiler. It could be trivially extended to cover the whole set of PL/I. The statement recognizer described in Chapter 3 has been tested on a number of test programs. On a 516 PL/I statements program, its execution time (GO step) was between 42.95 seconds and 48.27 seconds. The whole compiler program has more than 7300 cards. Its compilation time (EOL step) was between 251.12 seconds and 279.52 seconds.

A notable feature of the statement recognizer is the debugging facility. It was designed especially for pass 2 programmer. This mechanism is composed of procedures READB, DICTB, and DEBUG and the initial instructions at the beginning of the procedure STATEMENT.RECOGNIZER. If one would like to know the contents of the stack STRING after 'CALL READ' only, after 'CALL DICT' only or after both 'CALL READ' and 'CALL DICT', he may punch a '+', a '-' or an '=' respectively at the first column of a control card which should be the first card of his input data deck, i.e., the PL/I program deck.

Many valuable comments are also presented through the whole compiler program. The extensive compiler program listing occupies 138 pages and the result of a sample test program occupies 37 pages. They have too many pages to be included as an appendix.

BIBLIOGRAPHY

Barron, D. W., Recursive Techniques in Programming. American Elsevier, 1968.

Bates, F. and Douglas, M. L., Programming Language/One. Prentice-Hall, 1967.

Dijkstra, E. W., Making a Translator for Algol 60. A.P.I.C. Bulletin 7, pp. 3-11, 1961.

Freiburghouse, R. A., The Multics PL/I Compiler. AFIPS Conference Proceedings, Volume 35, 1969 Fall Joint Computer Conference. AFIPS Press, 1969.

Gear, C. W., Computer Organization and Programming. McGraw-Hill, 1969.

Grau, A. A., Hill, U. and Langmaack, H., Translation of Algol 60. Springer-Verlay, 1967.

Halstead, M. H., Machine-Independent Computer Programming. Spartan, 1962.

Hassitt, A., Computer Programming and Computer Systems. Academic Press, 1967.

Hopgood, F. R. A., Compiling Techniques. Macdonald/Elsevier, 1970.

IBM, PL/I Language Specifications. Form Y33-6003-1. April, 1969.

Ingermann, P. Z., A Syntax-Oriented Translator. Academic Press, 1966.

Lecht, C. P., The Programmer's PL/I - A Complete Reference. McGraw-Hill, 1968.

Lukaszewicz, L. and Nievergelt, J., EOL Programming Examples - A Primer. Report No. 242, Department of Computer Science, University of Illinois, Urbana, Illinois, September, 1967.

________, EOL Report. Report No. 241, Department of Computer Science, University of Illinois, Urbana, Illinois, September, 1967.

Pollack, S. V. and Sterling, T. D., A Guide to PL/I. Halt, Rinehart and Winston, 1969.

Randell, B. and Russell, L. J., Algol 60 Implementation. Academic Press, 1964.

Rosen, S., Programming Systems and Languages. McGraw-Hill, 1967.

Sprowls, R. C., Introduction to PL/I Programming. Harper and Row, 1969.

Wegner, P., Introduction to Systems Programming. Academic Press, 1964.

________, Programming Languages, Information Structures, and Machine Organization. McGraw-Hill, 1968.

APPENDIX A

THE TIMETABLE OF THE PROPOSED PL/I COMPILER

The following types of PL/I statements and attributes are desired to be implemented in the proposed PL/I compiler.

* denotes that the author has implemented them in his statement recognizer.

First Version

STATEMENTS:

* Assignment (scalars, arrays)
* BEGIN
* CALL (SUBROUTINE)
* DECLARE
* DISPLAY (REPLY)
* DO
* END
* EXIT
* GET (FILE--SKIP)
* GO TO (CONSTANT, VARIABLE)
* IF
* Null
* PROCEDURE (EXTERNAL)
* PUT (FILE)
* RETURN
* STOP

ATTRIBUTES:

AUTOMATIC, STATIC

BINARY, DECIMAL

CHARACTER

Dimension

ENTRY

EXTERNAL, INTERNAL

FILE

FIXED, FLOAT

INITIAL

INPUT, OUTPUT, UPDATE

LABEL

Length

OPTIONS

Parameter

Precision

PRINT

REAL

RETURNS

STREAM

VARIABLE

VARYING

<u>Second Version</u>

<u>STATEMENTS</u>:

* Assignment (structures, BY NAME)

CLOSE

* ENTRY

* FORMAT

* GET (FILE--COPY)

ON (SYSTEM)

OPEN

* PROCEDURE (INTERNAL)

<u>ATTRIBUTES</u>:

BIT

BITSTREAM

BUILTIN

COMPLEX

LIKE

<u>Third Version</u>

<u>STATEMENTS</u>:

* DEFAULT

DELETE

LOCATE

ON (SNAP)

READ

RELEASE

REVERT

REWRITE

SIGNAL

UNLOCK

WRITE

<u>ATTRIBUTES</u>:

BACKWARDS

BUFFERED, UNBUFFERED

CONNECTED

CONTROLLED

DEFINED

DIRECT

ENVIRONMENT

EVENT

GENERIC

IRREDUCIBLE, REDUCIBLE

KEYED

PICTURE

POSITION

RECORD

Fourth Version

STATEMENTS:

ALLOCATE

* BEGIN (ORDER/REORDER)

* CALL (TASK, EVENT, PRIORITY)

DELAY

* DISPLAY (EVENT)

FETCH

FREE

* GET (STRING, BITSTRING)

INCORPORATE

* PROCEDURE (RECURSIVE, ORDER/REORDER)

* PUT (STRING, BITSTRING)

WAIT

ATTRIBUTES:

ALIGNED, UNALIGNED

AREA

BASED

EXCLUSIVE

OFFSET, POINTER

SECONDARY

Size

TASK

APPENDIX B

WHEN TO CALL PROCEDURE DICT

The general rule is that the procedure DICT should be called whenever an identifier is encountered.

Assignment

Call DICT when an identifier is in top level of STRING stack.

BEGIN

Call DICT when BEGIN is in top level of STRING stack.

CALL

Call DICT when *EN

entry-name is in top two levels of STRING stack.

Call DICT when each name in an argument is in top level of STRING stack.

Call DICT when scalar-task-expression or scalar-event-expression or an identifier in scalar-expression is in top level of STRING stack.

DECLARE or DCL

Call DICT when DECLARE or DCL is in top level of STRING stack.

DEFAULT

Call DICT when DEFAULT is in top level of STRING stack.

DISPLAY

Call DICT when an identifier in scalar-expression is in top level of STRING stack.

Call DICT when scalar-character-variable or scalar-event-variable is in top level of STRING stack.

DO

Call DICT when DO is in top level of STRING stack.

Call DICT when an identifier is in top level of STRING stack.

END

Call DICT when END is in top level of STRING stack.

ENTRY

Call DICT when ENTRY is in top level of STRING stack.

Call DICT when each name in the parameter(s) is in top level of STRING stack.

<u>EXIT</u>

Do not call DICT.

<u>FORMAT</u>

Call DICT when each name in the format-list is in top level of STRING stack.

<u>GET</u>

Call DICT when *SIF

filename is in top two levels of STRING stack.

Call DICT when *CSN

first item in scalar-character-string-expression is in top two levels of STRING stack.

Call DICT when *BSN

first item in scalar-bit-string-expression is in top two levels of STRING stack.

Call DICT when each name in the data-specification or scalar-expression is in top level of STRING stack.

<u>GO TO or GOTO</u>

Call DICT when *L

label is in top two levels of STRING stack.

Call DICT when an identifier is in top level of STRING stack.

<u>IF</u>

Call DICT when each name in scalar-expression is in top level of STRING stack.

Call DICT whenever necessary in unit-1 and unit-2 according to what type of statement it is.

PROCEDURE or PROC

Call DICT when PROCEDURE or PROC is in top level of STRING stack.

Call DICT when each name in the parameter(s) is in top level of STRING stack.

PUT

Call DICT when *SOF

filename is in top two levels of STRING stack.

Call DICT when *CSN

scalar-character-string-variable is in top two levels of STRING stack.

Call DICT when *BSN

scalar-bit-string-variable is in top two levels of STRING stack.

Call DICT when each name in the data-specification or expression is in top level of STRING stack.

RETURN

Call DICT when RETURN is in top level of STRING stack.

Call DICT when an identifier in scalar-expression is in top level of STRING stack.

STOP

Do not call DICT.